Pointers, Virtual Functions and Polymorphism

9

Key Concepts

Polymorphism | Pointers | Pointers to objects | this pointer | Pointers to derived classes | Virtual functions | Pure virtual function

9.1 Introduction

Polymorphism is one of the crucial features of OOP. It simply means 'one name, multiple forms'. We have already seen how the concept of *polymorphism*is implemented using the overloaded functions and operators. The overloaded member functions are 'selected' for invoking by matching arguments, both type and number. This information is known to the compiler at the compile time and, therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. This is called *early binding*or *static binding*or *static linking.*Also known as *compile time polymorphism,*early binding simply means that an object is bound to its function call at compile time.

Now let us consider a situation where the function name and prototype is the same in both the base and derived classes. For example, consider the following class definitions:

```
class A
{
int x;
public:
void show()  {....}    // show() in base class
};
class B: public A
{
int y;
public:
void show()  {....}     // show() in derived class
};
```

How do we use the member function **show( )** to print the values of objects of both the classes **A** and **B?** Since the prototype of **show( )** is the same in both the places, the function is not overloaded and therefore static binding does not apply. We have seen earlier that, in such situations, we may use the class resolution operator to specify the class while invoking the functions with the derived class objects.

It would be nice if the appropriate member function could be selected while the program is running. This is known as *run time polymorphism.* How could it happen? C++ supports a mechanism known as *virtual function* to achieve run time polymorphism. Please refer Fig. 9.1.

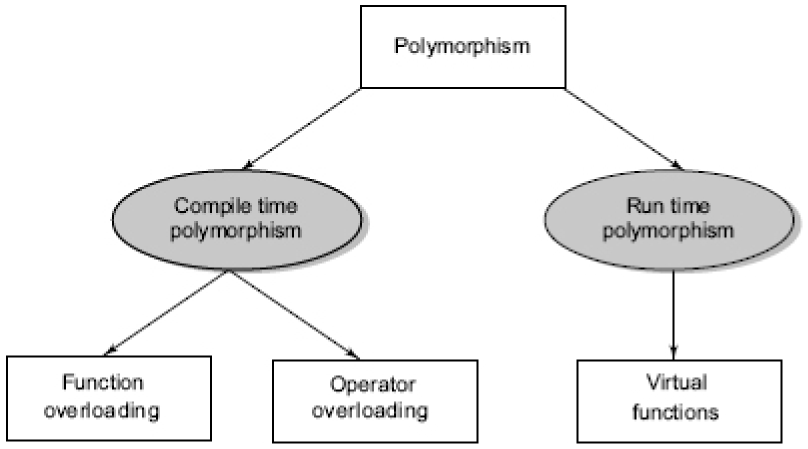

Fig. 9.1 *Achieving polymorphism*

At run time, when it is known what class objects are under consideration, the appropriate version of the function is invoked. Since the function is linked with a particular class much later after the compilation, this process is termed as *late binding.* It is also known as *dynamic binding* because the selection of the appropriate function is done dynamically at run time.

Dynamic binding is one of the powerful features of C++. This requires the use of pointers to objects. We shall discuss in detail how the object pointers and virtual functions are used to implement dynamic binding.

9.2 Pointers

Pointers is one of the key aspects of C++ language similar to that of C. As we know, pointers offer a unique approach to handle data in C and C++. We have seen some of the applications of pointers in Chapters 3 and 5. In this section, we shall discuss the rudiments of pointers and the special usage of them in C++.

We know that a pointer is a derived data type that refers to another data variable by storing the variable's memory address rather than data. A pointer variable defines where to get the value of a specific data variable instead of defining actual data.

Like C, a pointer variable can also refer to (or point to) another pointer in C++. However, it often points to a data variable. Pointers provide an alternative approach to access other data objects.

Declaring and Initializing Pointers

As discussed in Chapter 3, we can declare a pointer variable similar to other variables in C++. Like C, the declaration is based on the data type of the variable it points to. The declaration of a pointer variable takes the following form:

```
data-type *pointer-variable;
```

Here, *pointer-variable*is the name of the pointer, and the *data-type*refers to one of the valid C++ data types, such as int, char, float, and so on. The *data-type*is followed by an asterisk (*) symbol, which distinguishes a pointer variable from other variables to the compiler.

NOTE: *We can locate asterisk (*) immediately before the pointer variable, or between the data type and the pointer variable, or immediately after the data type. It does not cause any effect in the execution process.*

As we know, a pointer variable can point to any type of data available in C++. However, it is necessary to understand that a pointer is able to point to only one data type at the specific time. Let us declare a pointer variable, which points to an integer variable, as follows:

```
int  *ptr;
```

Here, **ptr** is a pointer variable and points to an integer data type. The pointer variable, ptr, should contain the memory location of any integer variable. In the same manner, we can declare pointer variables for other data types also.

Like other programming languages, a variable must be initialized before using it in a C++ program. We can initialize a pointer variable as follows:

```
int *ptr, a;   // declaration
ptr=&a;        // initialization
```

The pointer variable, **ptr,** contains the address of the variable **a.** Like C, we use the ‘address of’ operator or reference operator i.e., ‘&’ to retrieve the address of a variable. The second statement assigns the address of the variable **a** to the pointer **ptr.**

We can also declare a pointer variable to point to another pointer, similar to that of C. That is, a pointer variable contains address of another pointer. Program 9.1 explains how to refer to a pointer’s address by using a pointer in a C++ program.

Program 9.1 Example of Using Pointers

```
#include <iostream.h>
#include <conio.h>
void main( )
{
int a,  *ptr1,  **ptr2;
clrscr( );
ptr1 = &a;
ptr2=&ptr1;
cout << “The address of a : “ << ptrl << “\n”
cout << “The address of ptrl : “ << ptr2;
cout << “\n\n”;
cout << “After incrementing the address values:\n\n”;
ptr1+=2;
cout << “The address of a : “ << ptr1 << “\n”;
ptr2+=2;
cout << “The address of ptr1 : “ << ptr2 << “\n”;
}
```

The memory location is always addressed by the operating system. The output may vary depending on the system. Output of Program 9.1 would look like:

```
The address of a:      0x8fb6fff4
The address of ptr1:  0x8fb6fff2
After incrementing the address values:
The address of a: 0x8fb6fff8
The address of a:    0x8fb6fff6
```

We can also use *void pointers,*known as generic pointers, which refer to variables of any data type. Before using void pointers, we must type cast the variables to the specific data types that they point to.

NOTE: *The pointers, which are not initialized in a program, are called Null pointers. Pointers of any data type can be assigned with one value i.e., ‘0’ called null address.*

Manipulation of Pointers

As discussed earlier, we can manipulate a pointer with the indirection operator, i.e., ‘ ‘ which is also known as dereference operator. With this operator, we can indirectly access the data variable content. It takes the following general form:

```
*pointer_variable
```

As we know, dereferencing a pointer allows us to get the content of the memory location that the pointer points to. After assigning address of the variable to a pointer, we may want to change the content of the variable. Using the dereference operator, we can change the contents of the memory location.

Let us consider an example that illustrates how to dereference a pointer variable. The value associated with the memory address is divided by 2 using the dereference operator. The division affects only the memory contents and not the memory address itself. Program 9.2 illustrates the use of dereference operator in C++.

Program 9.2 Manipulation of Pointers

```
#include <iostream.h>
#include <conio.h>
void main( )
{
int a=10, *ptr;
```

```
        ptr - &a;
        clrscr( );
        cout << “The value of a is : “ << a;
        cout << “\n\n”;
        *ptr;-(*ptr;)/2
        cout << “The value of a is : “ << *ptr;
        cout << “\n\n”;
```

The output of Program 9.2 would be:

```
The value of a is : 10
The value of a is : 5
```

CAUTION: *Before dereferencing a pointer, it is essential to assign a value to the pointer. If we attempt to dereference an uninitialized pointer, it will cause runtime error by referring to any other location in memory.*

Pointer Expressions and Pointer Arithmetic

As discussed in Chapter 3, there are a substantial number of arithmetic operations that can be performed with pointers. C++ allows pointers to perform the following arithmetic operations:

A pointer can be incremented (++) (or) decremented (—)
Any integer can be added to or subtracted from a pointer
One pointer can be subtracted from another

Example:

```
int a[6];
int *aptr;
aptr=&a[0];
```

Obviously, the pointer variable, **aptr,** refers to the base address of the variable **a.** We can increment the pointer variable, shown as follows:

```
aptr++ (or) ++aptr
```

This statement moves the pointer to the next memory address. Similarly, we can decrement the pointer variable, as follows:

```
aptr− (or) −aptr
```

This statement moves the pointer to the previous memory address. Also, if two pointer variables point to the same array they can be subtracted from each other.

We cannot perform pointer arithmetic on variables which are not stored in contiguous memory locations. Program 9.3 illustrates the arithmetic operations that we can perform with pointers.

Program 9.3 Arithmetic Operations on Pointers

```
#include<iostream.h>
#include<conio.h>
void main()
{
int num[ ]={56,75,22,18,90};
int *ptr;
int i;
clrscr( );
cout << “The array values are:\n”;
for(i=0;i<5;i++)
   cout<< num[i]<<“\n”;
/*Allocating the base address of num to ptr*/
ptr = num;
/* Printing the value in the array */ cout <<“\nValue of ptr : “<<
*ptr; cout << “\n”; ptr++;
     cout<<“\nValue of ptr++ : “<< *ptr;
     cout << “\n”;
     ptr--;
     cout<<“\nValue of ptr— : “<< *ptr;
     cout << “\n”;
     ptr=ptr+2;
     cout<<“\nValue of ptr+2 : “<< *ptr;
     cout << “\n”;
     ptr−ptr-1;
     cout <<“\nValue of ptr-1: “<< *ptr;
     cout << “\n”;
     ptr+−3;
     cout<<“\nValue of ptr+=3: “<< *ptr;
     ptr-−2;
     cout << “\n”;
     cout<<“\nValue of ptr-=2: “<< *ptr;
     cout << “ \n”;
     getch( );
     }
```

The output of Program 9.3 would be:

```
The array values are:
56
75
22
18
90
Value of ptr    : 56
Value of ptr++  : 75
Value of ptr−−  : 56
Value of ptr+2  : 22
Value of ptr-1  : 75
Value of ptr+−3 : 90
Value of ptr-−2 : 22
```

Using Pointers with Arrays and Strings

Pointer is one of the efficient tools to access elements of an array. Pointers are useful to allocate arrays dynamically, i.e., we can decide the array size at run time. To achieve this, we use the functions, namely **malloc( )** and **calloc( ),** which we already discussed in Chapter 3. Accessing an array with pointers is simpler than accessing the array index.

In general, there are some differences between pointers and arrays; arrays refer to a block of memory space, whereas pointers do not refer to any section of memory. The memory addresses of arrays cannot be changed, whereas the content of the pointer variables, such as the memory addresses that it refers to, can be changed.

Even though there are subtle differences between pointers and arrays, they have a strong relationship between them.

NOTE: *There is no error checking of array bounds in C++. Suppose we declare an array of size 25. The compiler issues no warnings if we attempt to access 26th location. It is the programmer's task to check the array limits.*

We can declare the pointers to arrays as follows:

```
int *nptr;
nptr−number[0];
```

Or

```
nptr−number;
```

Here, **nptr** points to the first element of the integer array, number[0]. Also, consider the following example:

```
float *fptr;
fptr−price[0];
```

Or

```
fptr−price;
```

Here, **fptr** points to the first element of the array of float, price[0]. Program 9.4 demonstrates how linear search is performed in an array using pointers.

Program 9.4 Pointers with Arrays

```
#include<iostream>
#include<conio.h>
using namespace std;
void main( )
{
  int arr[10] − {1,99,4,37,88,3,19,45,62,87};
  int i,num,*ptr;
  /*Assigning the base address of array arr to ptr”*/
  ptr−arr;
  cout<<“Enter the element to be searched: ”;
  cin>>num;
  for(i=0;i<10;i++)
  {

if(*ptr−−num) {

cout<<“\n”<<num<<“ is present in the array”;

break;

}
else if(i−−9)

cout<<“\n”<<num<<“ is not present in the array”;

ptr++; //Incrementing the pointer to make it point

//to the next array element

        }
        getch( );
     }
```

The output of Program 9.4 would be:

```
Enter the element to be searched: 87
87 is present in the array
```

Arrays of Pointers

Similar to other variables, we can create an array of pointers in C++. The array of pointers represents a collection of addresses. By declaring array of pointers, we can save a substantial amount of memory space.

An array of pointers point to an array of data items. Each element of the pointer array points to an item of the data array. Data items can be accessed either directly or by dereferencing the elements of pointer array. We can reorganize the pointer elements without affecting the data items.

We can declare an array of pointers as follows:

```
int *inarray[10];
```

This statement declares an array of 10 pointers, each of which points to an integer. The address of the first pointer is inarray[0], and the second pointer is inarray[1], and the final pointer points to inarray[9]. Before initializing, they point to some unknown values in the memory space. We can use the pointer variable to refer to some specific values. Program 9.5 explains the implementation of array of pointers.

Program 9.5 Arrays of Pointers

```
#include <iostream.h>
#include <conio.h> #include <string.h>
#include <ctype.h> void main( )
{
int i−0;
char *ptr[10] − {
   “books”,
   “television”,
   “computer”,
   “sports”
   };
char str[25];
clrscr( );
cout << “\n\n\n\nEnter your favorite leisure pursuit: ” ;
cin >> str;
for(i=0; i<4; i++)
{
   if(!strcmp(str,  *ptr[i]))
   {
   cout << “\n\nYour favorite pursuit ” << “ is available
   here”
   << end1;
   break;
   }
}
if(i−−4)
cout << “\n\nYour favorite ” << “ not available here” <<
```

```
        endl;
        getch( );
        }
```

The output of Program 9.5 would be:

```
Enter your favorite leisure pursuit: books
Your favorite pursuit is available here
```

Pointers and Strings

We have seen the usage of pointers with one dimensional array elements. However, pointers are also efficient to access two dimensional and multi-dimensional arrays in C++. There is a definite relationship between arrays and pointers. C++ also allows us to handle the special kind of arrays, i.e., strings with pointers.

We know that a string is one dimensional array of characters, which start with the index 0 and ends with the null character ‘\0’ in C++. A pointer variable can access a string by referring to its first character. As we know, there are two ways to assign a value to a string. We can use the character array or variable of type char *. Let us consider the following string declarations:

```
char num[ ]=“one”;
const char *numptr− “one”;
```

The first declaration creates an array of four characters, which contains the characters, ‘o’,’n’,’e’,’\0’, whereas the second declaration generates a pointer variable, which points to the first character, i.e., ‘o’ of the string. There are numerous string handling functions available in C++. All of these functions are available in the header file <cstring>.

Program 9.6 demonstrates how to search a substring within a main string:

Program 9.6 Searching a Substring within a Main String

```
    #include<iostream>
    #include<conio.h>
    #include<string>
    using namespace std;
    void main( )
    {
    int i, j;
    char str[ ] – “\nC++ is better than C”;
    int len – strlen(str);
    char *substr – new char[len];
    cout<<“The main string is:  ”<<str;
    cout<<“\n\nEnter the substring to be searched:  ”;
    cin>>substr;
    int k,len2=strlen(substr);
    for(i=0;i<len;i++)
    {
      k=i;
      for(j=0;j<len2;j++)
      {

if(str[k]==substr[j])

{

if(j==len2-1) {

cout<<“\nThe substring is present in the main

string”;

getch( );
exit(0);

}

k++;

}

else

break;

        }
    }
    if(i==len)
    cout<<"\nThe substring is not present in the main string";
    getch( );
    }
```

The output of Program 9.6 would be:

```
The main string is:
C++ is better than C
Enter the substring to be searched: bet
The substring is present in the main string
```

Pointers to Functions

Even though pointers to functions (or function pointers) are introduced in C, they are widely used in C++ for dynamic binding, and event-based applications. The concept of pointer to function acts as a base for pointers to members, which we have discussed in Chapter 5.

The pointer to function is known as callback function. We can use these function pointers to refer to a function. Using function pointers, we can allow a C++ program to select a function dynamically at run time. We can also pass a function as an argument to another function. Here, the function is passed as a pointer. The function pointers cannot be dereferenced. C++ also allows us to compare two function pointers.

C++ provides two types of function pointers; function pointers that point to static member functions and function pointers that point to non-static member functions. These two function pointers are incompatible with each other. The function pointers that point to the non-static member function requires hidden argument.

Like other variables, we can declare a function pointer in C++. It takes the following form:

```
data_type(*function_name)( );
```

As we know, the data_type is any valid data types used in C++. The function_name is the name of a function, which must be preceded by an asterisk (*). The function_name is any valid name of the function.

Example:

```
int  (*num_function(int x));
```

Remember that declaring a pointer only creates a pointer. It does not create actual function. For this, we must define the task, which is to be performed by the function. The function must have the same return type and arguments. Program 9.7 explains how to declare and define function pointers in C++.

Program 9.7 Pointers to Functions

```
#include <iostream.h>
typedef void (*FunPtr) (int, int);
```

```
void Add&#16+0;(int i, int j)
{
cout << i << “ + “ << j << “ = “ << i + j;
}
void Subtract(int i, int j)
{
cout << i << “ - “ << j << “ = “ << i - j;
}
void main( )
{
FunPtr ptr;
ptr = &Add;
ptr(1,2);
cout << endl;
ptr = &Subtract;
ptr(3,2);
}
```

The output of Program 9.7 would be:

```
1 + 2 = 3
3 - 2 = 1
```

9.3 Pointers to Objects

We have already seen how to use pointers to access the class members. As stated earlier, a pointer can point to an object created by a class. Consider the following statement:

```
item x;
```

where **item** is a class and x is an object defined to be of type item. Similarly we can define a pointer **it_ptr** of type **item** as follows:

```
item *it_ptr;
```

Object pointers are useful in creating objects at run time. We can also use an object pointer to access the public members of an object. Consider a class **item** defined as follows:

```
class item
{
  int code;
  float price;
public:
void getdata(int a, float b)
  {

code = a;

price = b;

  }
  void show(void)
  {

cout << “Code : “ << code << “\n”;

<< “Price: “ << price

<< “\n\n”;
  }
};
```

Let us declare an **item** variable **x** and a pointer **ptr** to **x** as follows:

```
item x;
item *ptr = &x;
```

The pointer **ptr** is initialized with the address of **x.**

We can refer to the member functions of **item** in two ways, one by using the *dot operator*and *the object,*and another by using the *arrow operator*and the *object pointer.*The statements

```
x.getdata(100,75.50);
x.show( );
```

are equivalent to

```
ptr->getdata(100,  75.50);
ptr->show( );
```

Since ***ptr** is an alias of **x,** we can also use the following method:

```
(*ptr).show( );
```

The parentheses are necessary because the dot operator has higher precedence than the *indirection operator**.

We can also create the objects using pointers and **new** operator as follows:

```
item *ptr = new item;
```

This statement allocates enough memory for the data members in the object structure and assigns the address of the memory space to **ptr.** Then **ptr** can be used to refer to the members as shown below:

```
ptr -> show( );
```

If a class has a constructor with arguments and does not include an empty constructor, then we must supply the arguments when the object is created.

We can also create an array of objects using pointers. For example, the statement

```
item *ptr = new item[10];   // array of 10 objects
```

creates memory space for an array of 10 objects of **item.** Remember, in such cases, if the class contains constructors, it must also contain an empty constructor.

Program 9.8 illustrates the use of pointers to objects.

Program 9.8 Pointers to Objects

```
#include <iostream>
using namespace std;
class item
{
int code;
float price;
public:
void getdata(int a,  float b)
{
   code = a;
   price = b;
}
void show(void) {
   cout << “Code : “ << code << “\n”;
   cout << “Price: “ << price << “\n”;
}
};
const int size = 2;
int main( )
{
item *p = new item [size];
item *d = p;
int x, i;
float y;
for(i=0; i<size; i++)
{
   cout << “Input code and price for item” <<
   cin >> x >> y;
   p->getdata(x,y);
   p++;
}
for(i=0; i<size; i++)
{
   cout << “Item:” << i+1 << “\n”;
   d->show( );
   d++;
}
return 0;
}
```

The output of Program 9.8 would be:

```
Input code and price for item1 40 500
Input code and price for item2 50 600
Item:1
Code :      40
Price: 500
Item:2
Code :      50
Price: 600
```

In Program 9.8 we created space dynamically for two objects of equal size. But this may not be the case always. For example, the objects of a class that contain character strings would not be of the same size. In such cases, we can define an array of pointers to objects that can be used to access the individual objects. This is illustrated in Program 9.9.

Program 9.9 Array of Pointers to Objects

```
#include <iostream>
#include <cstring>
using namespace std;
```

```
    class city
    {
    protected:
       char *name;
       int len;
    public:
       city( )
       {
len = 0;
name = new char[len+1];         }
void getname(void)
{
char *s;
s = new char[30];
cout << “Enter city name:”;
cin >> s;
len = strlen(s);
name = new char[len + 1];
strcpy(name, s);
}
void printname(void)
{
cout << name << “\n”;
}      };      int main( )      {

city *cptr[10]; // array of 10 pointers to cities

int n = 1; int option;

do

{

cptr[n] = new city; // create new city

cptr[n]->getname( );

n++;

cout << “Do you want to enter one more name?\n”;

cout << “(Enter 1 for yes 0 for no):”;

cin >> option;

}

while(option);

cout << “\n\n”; for(int i=1; i<=n; i++) {

cptr[i]->printname( );

}

return 0;

    }
```

The output of Program 9.9 would be:

```
Enter city name:Hyderabad
Do you want to enter one more name?
(Enter 1 for yes 0 for no);1
Enter city name:Secunderabad
Do you want to enter one more name?
(Enter 1 for yes 0 for no);1
Enter city name:Malkajgiri
Do you want to enter one more name?
(Enter 1 for yes 0 for no);0

Hyderabad
Secunderabad
Malkajgiri
```

9.4 this Pointer

C++ uses a unique keyword called **this** to represent an object that invokes a member function. **this** is a pointer that points to the object for which *this*function was called. For example, the function call **A.max( )** will set the pointer **this** to the address of the object **A.** The starting address is the same as the address of the first variable in the class structure.

This unique pointer is automatically passed to a member function when it is called. The pointer **this** acts as an *implicit*argument to all the member functions. Consider the following simple example:

```
    class ABC
    {

int a;

. . . . .

. . . . .
};
```

The private variable ‘a’ can be used directly inside a member function, like

```
a = 123;
```

We can also use the following statement to do the same job:

```
this->a = 123;
```

Since C++ permits the use of shorthand form **a = 123,** we have not been using the pointer **this** explicitly so far. However, we have been implicitly using the pointer **this** when overloading the operators using member function.

Recall that, when a binary operator is overloaded using a member function, we pass only one argument to the function. The other argument is implicitly passed using the pointer **this.** One important application of the pointer **this** is to return the object it points to. For example, the statement

```
return *this;
```

inside a member function will return the object that invoked the function. This statement assumes importance when we want to compare two or more objects inside a member function and return the invoking object as a result. Example:

```
person & person :: greater(person & x)
{
if x.age > age
```

return x; // *argument object* else

return *this; // *invoking object* }

Suppose we invoke this function by the call

```
max = A.greater(B);
```

The function will return the object **B** (argument object) if the age of the person **B** is greater than that of **A,** otherwise, it will return the object **A** (invoking object) using the pointer **this.** Remember, the dereference operator * produces the contents at the address contained in the pointer. A complete program to illustrate the use of **this** is given in Program 9.10.

Program 9.10 this Pointer

```
#include <iostream>
#include <cstring>
using namespace std;
class person
{
char name[20];
float age;
public:
person(char *s, float a)
{
   strcpy(name,  s);
   age = a;
}
person & person :: greater(person & x)
{
   if(x.age >= age)

return x;         else

return *this;      }      void display(void)      {

cout << “Name: ” << name << “\n”
<< "Age: " << age << "\n";      }      };      int main( )      {
   person P1("John", 37.50),

P2("Ahmed", 2 9.0),

P3("Hebber", 40.25);

   person P = P1.greater(P3);    // P3.greater(P1)
   cout << "Elder person is: \n";
   P.display( );

   P = P1.greater(P2);        // P2.greater(P1)
   cout << "Elder person is: \n";
   P.display( );
   return 0;
}
```

The output of Program 9.10 would be:

```
Elder person is:
Name: Hebber
Age: 4 0.25
Elder person is:
Name: John
Age: 37.5
```

9.5 Pointers to Derived Classes

We can use pointers not only to the base objects but also to the objects of derived classes. Pointers to objects of a base class are type-compatible with pointers to objects of a derived class. Therefore, a single pointer variable can be made to point to objects belonging to different classes. For example, if **B** is a base class and **D** is a derived class from **B,** then a pointer declared as a pointer to **B** can also be a pointer to **D.** Consider the following declarations:

```
B *cptr;      //      pointer to class B type variable
B b;    //      base object
D d;    //      derived object
cptr = &b;   //      cptr points to object b
```

We can make **cptr** to point to the object **d** as follows:

```
cptr = &d;  // cptr points to object d
```

This is perfectly valid with C++ because **d** is an object derived from the class **B.**

However, there is a problem in using **cptr** to access the public members of the derived class **D.** Using **cptr,** we can access only those members which are inherited from **B** and not the members that originally belong to **D.** In case a member of **D** has the same name as one of the members of **B,** then any reference to that member by **cptr** will always access the base class member.

Although C++ permits a base pointer to point to any object derived from that base, the pointer cannot be directly used to access all the members of the derived class. We may have to use another pointer *declared*as pointer to the derived type.

Program 9.11 illustrates how pointers to a derived object are used.

Program 9.11 Pointers to Derived Objects

```
#include <iostream>
using namespace std;
class BC
{
public:
   int b;
   void show( )
```

```
  { cout << “b = ” << b << “\n”;}
};
class DC : public BC
{
public:
int d;
void show( )
{ cout << “b = ” << b << “\n”

<< “d = “ << d << “\n”;

}
};
int main( )
{
BC *bptr;   // base pointer
BC base;
bptr = &base;    // base address
bptr->b = 100;   // access BC via base pointer
cout << “bptr points to base object \n”;
bptr -> show( ); // derived class DC derived;
bptr = &derived;        // address of derived object
bptr -> b = 200; // access DC via base pointer

/* bptr -> d = 300;*/ // won’t work
cout << “bptr now points to derived object \n”;
bptr -> show( );       // bptr now points to derived object

/* accessing d using a pointer of type derived class DC */

DC *dptr;   // derived type pointer
dptr = &derived;
dptr->d = 300;

cout << “dptr is derived type pointer\n”; dptr -> show( );
cout << “using ((DC *)bptr)\n”;
((DC *)bptr)  -> d = 400; ((DC *)bptr)  -> show( );
```

```
      return 0;
}
```

The output of Program 9.11 would be:

```
bptr points base object
b = 100
bptr now points to derived object
b = 200
dptr is derived type pointer
b = 200
d = 300
using ((DC *)bptr)
b = 200
d = 400
```

NOTE: *We have used the statement*

```
bptr -> show( );
```

*two times. First, when **bptr** points to the base object, and second when **bptr** is made to point to the derived object. But, both the times, it executed **BC::show( )** function and displayed the content of the base object. However, the statements*

```
dptr -> show( );
((DC *) bptr)  -> show( );    // cast bptr to DC type
```

*display the contents of the **derived** object. This shows that, although a base pointer can be made to point to any number of derived objects, it cannot directly access the members defined by a derived class.*

9.6 Virtual Functions

As mentioned earlier, polymorphism refers to the property by which objects belonging to different classes are able to respond to the same message, but in different forms. An essential requirement of polymorphism is therefore the ability to refer to objects without any regard to their classes. This necessitates the use of a single pointer variable to refer to the objects of different classes. Here, we use the pointer to base class to refer to all the derived objects. But, we just discovered that a base pointer, even when it is made to contain the address of a derived class, always executes the function in the base class. The compiler simply ignores the contents of the pointer and chooses the member function that matches the type of the pointer. How do we then achieve polymorphism? It is achieved using what is known as 'virtual' functions.

When we use the same function name in both the base and derived classes, the function in base class is declared as *virtual*using the keyword **virtual** preceding its normal declaration. When a function is made **virtual, C++** determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer. Thus, by making the base pointer to point to different objects, we can execute different versions of the **virtual** function. Program 9.12 illustrates this point.

Program 9.12 Virtual Functions

```
#include <iostream>
using namespace std;
class Base
{
public:
   void display( )  {cout << "\n Display base ";} virtual void
```

```
show( )  {cout << “\n show base”;}
      class Derived : public Base
      {
      public:
         void display( )  {cout << “\n Display derived”;}
         void show( )  {cout << “\n show derived”;}
      };
      int main( )
      {
         Base B;
         Derived D;
         Base *bptr;

         cout << “\n bptr points to Base \n”; bptr = &B;
         bptr -> display( );    // calls Base version
         bptr -> show( );       // calls Base version

         cout << “\n\n bptr points to Derived\n”;
         bptr = &D;
         bptr -> display( );    // calls Base version
         bptr -> show( );       // calls Derived version

         return 0;
      }
```

The output of Program 9.12 would be:

```
bptr points to Base

Display
base Show base

bptr points to Derived

Display base
Show derived
```

NOTE: *When **bptr** is made to point to the object **D,** the statement*

```
bptr -> display( );
```

*calls only the function associated with the **Base** (i.e. **Base :: display()**), whereas the statement*

```
bptr -> show( );
```

*calls the **Derived** version of **show( ).** This is because the function **display( )** has not been made **virtual** in the**Base class.***

One important point to remember is that, we must access **virtual** functions through the use of a pointer declared as a pointer to the base class. Why can't we use the object name (with the dot operator) the same way as any other member function to call the virtual functions? We can, but remember, run time polymorphism is achieved only when a virtual function is accessed through a pointer to the base class.

Let us take an example where **virtual** functions are implemented in practice. Consider a book shop which sells both books and video-tapes. We can create a class known as **media** that stores the title and price of a publication. We can then create two derived classes, one for storing the number of pages in a book and another for storing the playing time of a tape. Figure 9.2 shows the class hierarchy for the book shop.

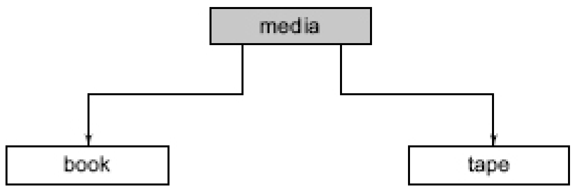

Fig. 9.2 *The class hierarchy for the book shop*

The classes are implemented in Program 9.13. A function **display( )** is used in all the classes to display the class contents. Notice that the function **display( )** has been declared virtual in media, the base class.

In the **main** program we create a heterogeneous list of pointers of type **media** as shown below:

```
media *list[2] = { &book1,  &tape1};
```

The base pointers **list[0]** and **list[1]** are initialized with the addresses of objects **book1** and **tape1** respectively.

Program 9.13 Runtime Polymorphism

```
#include <iostream>
#include <cstring>
using namespace std;
class media
{
protected:
   char title[50];
   float price;
public:
   media(char *s, float a)
   {

strcpy(title, s);

price = a;         }         virtual void display( ) { } // empty virtual
function      }      class book: public media      {         int pages;
      public:         book(char *s, float a, int p):media(s,a)         {

pages = p;         }         void display( );      };      class tape :public
media      {         float time;      public:         tape(char * s, float a,
float t):media(s, a)          {

time = t;          }          void display( );       };       void book :: display(
)       {          cout << “\n Title: ” << title;          cout << “\n Pages: ”
<< pages;          cout << “\n Price: ” << price;       }       void tape ::
display( )       {          cout << “\n Title: ” << title;          cout << “\n
play time: ” << time << “mins”;          cout << “\n price: ” << price;
     }       int main( )       {          char * title = new char[30];
        float price, time;          int pages;
        // Book details          cout << “\n ENTER BOOK DETAILS\n”;
        cout << “ Title: ”; cin >> title;          cout << “ Price: ”; cin >>
price;          cout << “ Pages: ”; cin >> pages;
        book book1(title, price, pages);
        // Tape details          cout << “\n ENTER TAPE DETAILS\n”;
        cout << “ Title: ”; cin >> title;          cout << “ Price: ”; cin >>
price;          cout << “ Play time (mins): ”; cin >> time;          tape
tape1(title, price, time);
        media* list[2];          list[0] = &book1;          list[1] = &tape1;
        cout << “\n MEDIA DETAILS”;
        cout << “\n......  BOOK  .......”;          list[0] -> display( ); //
display book details
        cout << “\n  .......TAPE  .......”;
        list[1] -> display( ); // display tape details

        result 0;
     }
```

The output of Program 9.13 would be:

```
ENTER BOOK DETAILS
Title:Programming_in_ANSI_C
Price: 88
Pages: 400

ENTER TAPE DETAILS
Title: Computing_Concepts
Price: 90
```

```
Play time  (mins): 55

MEDIA DETAILS
......  BOOK  ........
Title:Programming_in_ANSI_C
Pages: 400
Price: 88

......  TAPE  ........
Title: Computing_Concepts
Play time: 55mins
Price: 90
```

Rules for Virtual Functions

When virtual functions are created for implementing late binding, we should observe some basic rules that satisfy the compiler requirements:

1. The virtual functions must be members of some class.
2. They cannot be static members.
3. They are accessed by using object pointers.
4. A virtual function can be a friend of another class.
5. A virtual function in a base class must be defined, even though it may not be used.
6. The prototypes of the base class version of a virtual function and all the derived class versions must be identical. If two functions with the same name have different prototypes, C++ considers them as overloaded functions, and the virtual function mechanism is ignored.
7. We cannot have virtual constructors, but we can have virtual destructors.
8. While a base pointer can point to any type of the derived object, the reverse is not true. That is to say, we cannot use a pointer to a derived class to access an object of the base type.
9. When a base pointer points to a derived class, incrementing or decrementing it will not make it to point to the next object of the

derived class. It is incremented or decremented only relative to its base type. Therefore, we should not use this method to move the pointer to the next object.

10. If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, calls will invoke the base function.

9.7 Pure Virtual Functions

It is normal practice to declare a function virtual inside the base class and redefine it in the derived classes. The function inside the base class is seldom used for performing any task. It only serves as a *placeholder.*For example, we have not defined any object of class media and therefore the function display( ) in the base class has been defined 'empty'. Such functions are called "do-nothing" functions.

A "do-nothing" function may be defined as follows:

```
virtual void display( ) = 0;
```

Such functions are called *pure virtual*functions. A pure virtual function is a function declared in a base class that has no definition relative to the base class. In such cases, the compiler requires each derived class to either define the function or redeclare it as a pure virtual function. Remember that a class containing pure virtual functions cannot be used to declare any objects of its own. As stated earlier, such classes are called *abstract base classes.*The main objective of an abstract base class is to provide some traits to the derived classes and to create a base pointer required for achieving run time polymorphism.

Program 9.14 demonstrates the use of pure virtual function:

Program 9.14 Pure Virtual Function

```
     #include <iostream>
     using namespace std;
     class Balagurusamy  //base class
     {
     public:
     virtual void example( )=0;  //Denotes pure virtual Function
Definition
     };
     class C:public Balagurusamy    //derived class 1
     {
     public:

void example( ) {

cout<<“C text Book written by Balagurusamy”;

}      };      class oops:public Balagurusamy //derived class 2      {
     public:

void example( )

{

cout<<“C++ text Book written by Balagurusamy”;

}

     };
     void main( )
     {
     Exforsys* arra[2];
     C e1;
     oops e2;
     arra[0]=&e1;
```

```
    arra[1]=&e2;
    arra[0]->example( );
    arra[1]->example( );
    }
```

The output of Program 9.14 would be:

```
C text Book written by Balagurusamy
C++ text Book written by Balagurusamy
```

9.8 Virtual Constructors and Destructors

"A constructor can not be virtual". There are some valid reasons that justify this statement. First, to create an object the constructor of the object class must be of the same type as the class. But, this is not possible with a virtually implemented constructor. Second, at the time of calling a constructor, the virtual table would not have been created to resolve any virtual function calls. Thus, a virtual constructor itself would not have anywhere to look up to. As a result, it is not possible to declare a constructor as virtual.

A virtual destructor however, is pretty much feasible in C++. In fact, its use is often promoted in certain situations. One such situation is when we need to make sure that the different destructors in an inheritance hierarchy are called in order, particularly when the base class pointer is referring to a derived type object. It must be noted here that the order of calling of destructors in an inheritance hierarchy is opposite to that of constructors.

Consider the following class declarations that make use of destructors:

```
class A
{
public:
        ~A( )
        {

//Base class destructor          }      };      class B : public A      {
     public:          ~B( )          {

//Derived class destructor          }      };      main( )      {          A *ptr =
new B( );

        .
        .
        delete ptr;
     }
```

In the above class declarations, both base class A and the derived class B have their own destructors. Now, an A class pointer has been allocated a B class object. When this object pointer is deleted using the delete operator, it will trigger the base class destructor and the derived class destructor won't be called at all. This may lead to a memory leak situation. To make sure that the derived class destructor is mandatorily called, we must declare the base class destructor as virtual, as shown below.

```
     class A
     {
        public:
        virtual ~A( )
        {
//Base class destructor
        }
     };
     .
     .
```

Summary

- ❑ Polymorphism simply means one name having multiple forms.
- ❑ There are two types of polymorphism, namely, compile time polymorphism and run time polymorphism.
- ❑ Functions and operators overloading are examples of compile time polymorphism. The overloaded member functions are selected for invoking by matching arguments, both type and number. The compiler knows this information at the compile time and, therefore, compiler is able to select the appropriate function for a particular call at the compile time itself. This is called early or static binding or static linking. It means that an object is bound to its function call at compile time.
- ❑ In run time polymorphism, an appropriate member function is selected while the program is running. C++ supports run time polymorphism with the help of virtual functions. It is called late or dynamic binding because the appropriate function is selected dynamically at run time. Dynamic binding requires use of pointers to objects and is one of the powerful features of C++.
- ❑ Object pointers are useful in creating objects at run time. It can be used to access the public members of an object, along with an arrow operator.
- ❑ A **this** pointer refers to an object that currently invokes a member function. For example, the function call **a.show( )** will set the pointer ‘this’ to the address of the object ‘a’.
- ❑ Pointers to objects of a base class type are compatible with pointers to objects of a derived class. Therefore, we can use a single pointer variable to point to objects of base class as well as derived classes.

- ❑ When a function is made virtual, C++ determines which function to use at run time based on the type of object pointed to by the base pointer, rather than the type of the pointer. By making the base pointer to point to different objects, we can execute different versions of the virtual function.
- ❑ Run time polymorphism is achieved only when a virtual function is accessed through a pointer to the base class. It cannot be achieved using object name along with the dot operator to access virtual function.
- ❑ We can have virtual destructors but not virtual constructors.
- ❑ If a virtual function is defined in the base class, it need not be necessarily redefined in the derived class. In such cases, the respective calls will invoke the base class function.
- ❑ A virtual function, equated to zero is called a pure virtual function. It is a function declared in a base class that has no definition relative to the base class. A class containing such pure function is called an abstract class.

Key Terms

abstract base classes | ‘address of’ operator | argument object | arrays of pointers | arrow operator | base address | base object | base pointer | call back function |class hierarchy | compile time | compile time polymorphism | dereference operator |derived object | do-nothing function | dot operator | dynamic binding | early binding | function overloading | function pointer | implicit argument | indirection operator |invoking object | late binding | new operator | null pointers | object pointer | operator overloading | placeholder | pointers | pointer arithmetic | pointers to functions |polymorphism | pure virtual function | run time | run time polymorphism | static binding | static linking | this pointer | virtual constructors | virtual destructors | virtual function | void pointers

Review Questions

9.1 What does polymorphism mean in C++ language?

9.2 How is polymorphism achieved at (a) compile time, and (b) run time?

9.3 Discuss the different ways by which we can access public member functions of an object.

9.4 What is pointer to a function? Explain the situations in which such a concept is used.

9.5 Explain, with an example, how you would create space for an array of objects using pointers.

9.6 What does **this** pointer point to?

9.7 What are the applications of **this** pointer?

9.8 Can a pointer of base class type point to an object of the derived class? Explain.

9.9 What is a virtual function?

9.10 Why do we need virtual functions?

9.11 When do we make a virtual function "pure"? What are the implications of making a function a pure virtual function?

9.12 State which of the following statements are TRUE or FALSE.

(a) Virtual functions are used to create pointers to base classes.

(b) Virtual functions allow us to use the same function call to invoke member functions of objects of different classes.

(c) A pointer to a base class cannot be made to point to objects of derived class.

(d) this pointer points to the object that is currently used to invoke a function.

(e) this pointer can be used like any other pointer to access the members of the object it points to.

(f) this pointer can be made to point to any object by assigning the address of the object.

(g) Pure virtual functions force the programmer to redefine the virtual function inside the derived classes.

Debugging Exercises

9.1 Identify the error in the following program.

```
    #include <iostream.h>
    class Info
    {
      char *name;
      int number;
    public:
      void getInfo( )
      {

cout << “Info::getInfo ”;

getName( );        }      void getName( )      {

cout << “Info::getName ”;         }      };      class Name: public Info
    {        char *name;      public:        void getName( )        {

cout << “Name::getName ”;         }      };      void main( )      {
      Info *p;
      Name n;
      p = n;
      p->getInfo( );
    }
    /*
```

9.2 Identify the error in the following program.

```
  #include <iostream.h>
  class Person
  {
     int age;
  public:
     Person( )
     {
     }
     Person(int age)
     {
this.age = age;

     }
     Person& operator <  (Person &p)
     {
return age < p.age ? p: *this;
     }
     int getAge( )
     {
return age;
     }
  };
  Void main  ( )
  {
     Person P1 (15);
     Person P2 (11);
     Person P3;
     //if p1 is less than p2
```

```
        p3 = p1 < p2; p1. lessthan(p2)
        cout << p3.getAge( );
    }
    /*
```

9.3 Identify the error in the following program.

```
    #include “iostream.h”
    class Human
    {
    public:
        Human( )
        {
        }
        virtual ~Human( )
        {
cout << “Human::~Human”;
        }
    };
    class Student: public Human
    {
    public:
        Student( )
        {
        }
        ~Student( )
        {
cout << “Student::~Student( )”;
        }
    };
    void main( )
    {
        Human *H = new Student( );
        delete H;
    }
```

9.4 Correct the errors in the following program.

```
    class test
    {
    private:
       int m;
    public:
       void getdata( )
       {
cout <<“Enter number:”;
cin >> m;
       }
       void display( )
       {
cout << m;
       }
    };
    main( )
    {
       test T;
       T->getdata( );
       T->display( );
       test *p;
       p = new test;
       p.getdata( );
       (*p).display( );
    }
```

9.5 Debug and run the following program. What will be the output?

```
    #include <iostream.h>
    class A
    {
    protected:
       int a,b;
    public:
       A(int x = 0, int y)
       {
```

a = x;

b = y; } virtual void print( ); }; class B: public A { private: float p,q; public: B(int m, int n, float u, float v) {

p = u;

q = v; } B( ) {p = q = 0;} void input(float u, float v); virtual void print(float); }; void A::print(void) { cout << A values: << a <<" "<< b <<"\n"; } void B::print(float) { cout <<B values:<< u <<" "<< v <<"\n"; } void B::input(float x, float y) { p = x; q = y; } main( ) A a1(10,20), *ptr;

```
    B b1;
    b1.input(7.5,3.142);

    ptr = &a1;
    ptr->print( );

    ptr = &b1;
    ptr->print( );
  }
```

9.6 Test the output of the following program:

```
#include<iostream>
using namespace std;
class BC
{
public:
  void show(void){cout<<"\nI am in base class..";}
};
class DC: public BC
{
public:
  void show(void){cout<<"\nI am in derived class..";}
};
void main( )
{
  BC *ptr;
  DC dobj;
  ptr=&dobj;
  ptr->show( );
}
```

What would you change in the above program to trigger the show( ) method of the derived class instead of the base class?

Programming Exercises

9.1 Create a base class called **shape.** Use this class to store two **double** type values that could be used to compute the area of figures. Derive two specific classes called **triangle** and **rectangle** from the base **shape.** Add to the base class, a member function **get_data( )** to initialize base class data members and another member function **display_area( )** to compute and display the area of figures. Make **display_area( )** as a virtual function and redefine this function in the derived classes to suit their requirements.

Using these three classes, design a program that will accept dimensions of a triangle or a rect angle interactively, and display the area.

Remember the two values given as input will be treated as lengths of two sides in the case of rectangles, and as base and height in the case of triangles, and used as follows:

Area of rectangle = x * y
Area of triangle = 1/2 * x * y

9.2 Extend the above program to display the area of circles. This requires addition of a new derived class 'circle' that computes the area of a circle. Remember, for a circle we need only one value, its radius, but the get_data( ) function in the base class requires two values to be passed. (Hint: Make the second argument of get_data( ) function as a default one with zero value.)

9.3 Run the above program with the following modifications:

(a) Remove the definition of **display_area( )** from one of the derived classes.

(b) In addition to the above change, declare the **display_area( )** as **virtual** in the base class **shape.**

Comment on the output in each case.

9.4 Using the concept of pointers write a function that swaps the private data values of two objects of the same class type.

WEB